LE "CHAUDRON MAGIQUE" ET "L'INVENTION COLLECTIVE"

Pierre-André Mangolte

▶ **To cite this version:**

Pierre-André Mangolte. LE "CHAUDRON MAGIQUE" ET "L'INVENTION COLLECTIVE". Économie appliquée : archives de l'Institut de science économique appliquée, 2005, tome LVIII, 2005, n°1, pp.59-83. hal-00129395

HAL Id: hal-00129395
https://hal.science/hal-00129395

Submitted on 7 Feb 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

LE « CHAUDRON MAGIQUE » ET « L'INVENTION COLLECTIVE »

Pierre-André MANGOLTE

CEPN (IIDE) - CNRS UMR n° 7115
Université PARIS-NORD
99, Av. Jean-Baptiste Clément
93430 VILLETANEUSE – France

Septembre 2003 – Janvier 2005

e-mail : p.a.mangolte@wanadoo.fr

Résumé :

Cet article traite des logiciels libres et des projets de développement *open source*, c'est-à-dire de l'apparition d'une forme de production non-marchande du logiciel. Sans paiement ni investissement financier important, sans planning ni design préalable, des produits aussi complexes que Linux ou Apache ont pu voir le jour, comme s'ils sortaient d'une sorte de « chaudron magique » (Raymond, 1999). L'article cherche à comprendre et à dissiper les mystères de ce chaudron magique, en s'intéressant aux contraintes techniques de la production et au cycle de développement et de vie des logiciels; ces données étant rapprochées du cadre institutionnel existant (le *copyright* ou les licences *open source* (GPL, etc.)). Les formes d'organisation et de coordination de ces projets reposent alors sur la modularité, une certaine division du travail et un contrôle sélectif et hiérarchisé des différentes contributions. Ils s'inscrivent dans un schéma général "d'invention collective", où le libre partage et usage du code permet la production et l'évolution incrémentale des programmes.

Abstract :

This paper deals with free software and open source development projects, ie with the emergence of a form of noncommercial production of the software. Without payment nor important financial investment, without planification nor prior design, complex products as Linux or Apache could be born. All occurs as if they emerged from a kind of « magic cauldron » (Raymond, 1999). We try to understand and to clear up the mysteries of this « cauldron ». The production engineering constraints and the cycle of development and life of software are studied in the light of existing institutional framework (the copyright or open sources licences like GPL). The structures of organization and coordination of the open source projects rest then on the modularity, the emergence of division of labour, and a selective and hierarchical control of the various contributions. The evolution of this projects follows the general pattern of « collective invention » where the free sharing and use of the code allow production and incremental evolutions of the programs.

INTRODUCTION

L'importance grandissante des logiciels libres a, dans une période récente, attiré l'attention des économistes. Certains de ces logiciels libres menacent en effet les positions des logiciels commerciaux, ou dominent déjà leur propre « marché ». C'est le cas de Sendmail (un programme de transfert d'e-mail) par exemple, ou d'Apache qui aujourd'hui fait tourner la plupart des serveurs web dans le monde. Le système d'exploitation Linux connaît lui aussi un succès et une diffusion étonnante.

Le plus intrigant et le plus déroutant est sans doute pour l'économiste l'existence des « projets *open source* », une manière particulière d'organiser la production d'un logiciel. Il s'agit de formes de coopération volontaire entre des programmeurs dispersés un peu partout dans le monde, sans paiement ni échange monétaire, sans investissement financier important, alors que tout cela est nécessaire pour produire des logiciels dans des formes d'organisation plus traditionnelles. Pour certains, il s'agit de nouveautés radicales, car des logiciels de haute qualité semblent se matérialiser à partir de rien, comme s'ils sortaient d'un « chaudron magique ». C'est l'opinion de Raymond [1998 et 1999] en particulier, qui oppose ce mode anarchique de coopération entre pairs (le « bazar ») au mode traditionnel centralisé suivi jusqu'ici (la « cathédrale »). Cette thèse est cependant assez largement remise en cause et nuancée aujourd'hui [Bezroukhov (1999a, 1999b); Kuwabara (2000); Narduzzo et Rossi (2003)].

L'existence d'un tel chaudron magique rappelle à l'économiste que la production et l'utilisation des richesses articulent de manière assez compliquée parfois une sphère marchande et une sphère non-marchande. Appliquer à la production non-marchande les catégories et représentations issues de la micro-économie standard peut s'avérer alors problématique. Ainsi, quand on postule une fonction de production, dont l'output dépend du niveau des investissements, lesquels sont reliés à une quelconque espérance de profit, et qu'on applique ce concept au logiciel, on tend à justifier, comme un mal nécessaire et par l'effet d'un raisonnement à l'envers, les pratiques propriétaires de non divulgation du code et de restrictions en matière de copie et d'usage. Ce qui rend d'autant plus mystérieux l'existence et les succès des projets *open source*, puisque l'incitation qui découle de la recherche d'un gain monétaire n'existe pas. On ne peut de même poser les logiciels comme « des biens d'information », lesquels sont « chers à produire, mais pas à reproduire » [Shapiro et Varian (1999)]. Cette formule simple (et inexacte) évacue en effet tout ce que l'analyse devrait prendre en compte : (a) la difficulté de définir le logiciel comme un « bien », alors que le

classement comme « bien » ou « service » a toujours posé problème ici [Dréan (1996)], (b) la nature complexe de l'objet technique, une valeur d'usage bien particulière, et les multiples contraintes qui pèsent sur sa production, son développement, et son utilisation, (c) le cadre institutionnel, en termes de droits de propriété, un point qui justement sépare les logiciels libres et les logiciels commerciaux (ou « propriétaires »).

Car tout commence avec une innovation institutionnelle, une redéfinition des droits de propriété sur le logiciel. Le cadre juridique existant - *copyright* ou droit d'auteur - limitant fortement les pratiques de libre circulation des programmes, on a vu apparaître des licences *open source* (GPL, etc.), ouvrant la possibilité d'un processus ouvert « d'invention collective », avec les trois caractéristiques retenues par Allen [1983] : échange libre de connaissances, absence d'investissements spécifiques, non-appropriabilité des produits de "l'invention collective". Les nouvelles licences fixent des normes et encadrent les comportements, et c'est dans ce cadre qu'ont émergé les grands projets *open source* du chaudron magique, un chaudron magique dont nous voudrions tenter l'analyse.

Dans une première partie, nous étudierons les licences *open source* (comme la GNU-GPL), rappellerons leurs origines historiques et analyserons le pourquoi de cette innovation institutionnelle.

Dans une deuxième partie, nous traiterons du procès de développement du logiciel. On essayera d'identifier en particulier les « déterminismes *soft* » (Rosenberg, 1982), d'origine technologique, qui pèsent sur la production d'un logiciel. On analysera ensuite la manière dont les grands projets *open source* ont pu progressivement donner naissance à des formes spécifiques de coordination reposant sur la modularité, une certaine division du travail, et un contrôle sélectif des contributions.

I. LA BASE INSTITUTIONNELLE DU MAINTIEN DE « L'INVENTION COLLECTIVE »

Un logiciel est un ensemble d'instructions destiné à une machine qui prend la forme d'un texte écrit dans un langage de programmation (le code-source) ; ce texte est ensuite transformé (au moyen d'un compilateur) en code-objet (les instructions en langage machine). Le logiciel est une forme d'écriture, ce qui a conduit les juristes à classer les logiciels dans le cadre du *copyright* (ou du droit d'auteur). La commercialisation, quand elle existe, prend la forme d'une licence. Dans les licences commerciales habituelles (dites « propriétaires »), on interdit en général toute copie, toute redistribution ou modification du programme, lequel est

fourni sans accès direct au code-source[1]. C'est en réagissant contre l'apparition de ce type de licences que Richard Stallman et la *Free Software Foundation* ont lancé dès le milieu des années 1980 le mouvement des logiciels libres.

Dans les années 1970 en effet, AT&T (les Bells Lab) ne pouvait, pour cause d'anti-trust, développer aucune activité commerciale en informatique. C'est dans ce cadre non-marchand que furent conçus les premiers systèmes d'exploitation de type Unix, « par un petit groupe de personnes n'ayant qu'un but à l'esprit : élaborer un environnement idéal pour le développement du logiciel » [Gnoff et Weinberg (1989)][2]. Des pratiques de distribution libre des programmes (y compris pour des versions en cours de développement) et de partage des modifications, se mettent alors en place entre les Bell Labs et différentes universités (Berkeley en particulier). Mais ce système ouvert d'invention collective est par la suite déstabilisé par l'apparition de différents systèmes Unix propriétaires[3], le programme n'étant plus fourni que sous forme exécutable (sans le code-source), ou accompagné d'interfaces le rendant incompatibles avec d'autres programmes développés séparément. C'est pour contrer cette évolution, pour maintenir les pratiques du partage et la possibilité d'une production collective, qu'apparurent alors les premières licences *open source* (BSD et GPL).

« Quand des ingénieurs souhaitaient utiliser l'un de nos programmes, nous leur en accordions volontiers l'autorisation. Et quand on voyait quelqu'un utiliser un programme intéressant mais inconnu, on pouvait toujours en obtenir le code-source, afin de le lire, de le modifier, ou d'en vampiriser des parties dans le cadre d'un nouveau programme » [Stallman (1998a)]. Mais l'essor des logiciels propriétaires conduisait à un monde où « toute communauté coopérative serait interdite », où des murs de plus en plus hauts sépareraient les différents programmeurs. C'est ce qui conduisit Stallman à fonder la *Free Software Foundation*, à définir un nouveau type de licence (la licence GPL), et à lancer le projet GNU.

---

[1] Le système juridique actuel interdit sauf cas particuliers la décompilation du code-objet, une opération qui ne permet d'ailleurs pas de remonter jusqu'au code-source. On ne peut donc ré-écrire le code, même pour corriger des erreurs, ...

[2] Unix est issu d'un programme regroupant les Bell Labs, le MIT et l'ARPA (avec des financements fédéraux). Il s'agissait de construire un système portable (utilisable, après compilation, sur un parc de machines hétérogènes), multi-tâches, et multi-utilisateurs. La première version documentée date de 1971, et fut disponible en langage C (lequel fut élaboré parallèlement) en 1973 [Salus (1994)].

[3] L'apparition des trajectoires propriétaires concurrentes oblige alors à ré-écrire sans aucune nécessité technique le code ligne à ligne pour le libérer - cas des distributions BSD [Unix Berkeley (1999)]; et, en sens inverse, à le ré-écrire de la même façon pour le rendre "propriétaire". Le régime des droits de propriété démultiplie alors les coûts de production.

Les licences *open source*, une (re)définition des droits de propriété sur le logiciel

Dans la licence GNU-GPL, comme dans les autres licences *open source software*[4], le propriétaire des lignes de code accorde un droit d'usage particulièrement étendu à l'utilisateur. Celui-ci peut copier librement le programme, le distribuer à sa guise (moyennant paiement ou non), et le modifier comme il l'entend. Il obtient pour cela un libre accès au code-source. Ces droits distinguent toutes ces licences des licences commerciales habituelles.

Le logiciel n'est en effet pas traité comme un produit commercial dont il faut protéger le code et interdire toute copie, sous peine de pertes de recettes; mais comme une valeur d'usage, avec deux usages possibles du logiciel et deux sortes d'utilisateurs. Il y a l'utilisateur final essentiellement intéressé par les fonctionnalités du programme et la version code-objet qui peut tourner effectivement sur son propre système informatique, et les utilisateurs-programmeurs qui veulent pouvoir utiliser ce logiciel pour en produire un autre, en modifiant le code-source[5]. Il s'agit par exemple de réaliser une simple amélioration (correction des bogues), ou d'ajouter des extensions, de transférer le programme sur un autre système informatique, etc. Il arrive même que le programme soit considéré comme un ensemble de pièces ou d'éléments réutilisables pour développer un programme complètement différent du premier. Les licences *open source* sont faites pour ces deux catégories d'utilisateurs, et plus particulièrement pour la deuxième. Les droits concédés par la licence dépassent en effet largement le simple droit d'exécuter un programme sans le modifier, partant de l'idée que les programmes ont vocation a être utilisés, ré-utilisés et développés par leurs utilisateurs, lesquels doivent pouvoir pour cela accéder librement au code-source.

Pour assurer le développement d'un système de logiciels libres, il fallait instituer des conditions de distribution qui interdisent la transformation d'un logiciel libre en logiciel propriétaire. Car la mise volontaire d'un logiciel dans le domaine public ne suffit pas à empêcher sa privatisation éventuelle. Si le producteur du code a renoncé à faire valoir son droit d'auteur, le code n'appartient à personne, et peut assez facilement passer du domaine public au domaine des logiciels propriétaires. La GPL définit donc précisément qui est propriétaire du code, impose la redistribution de celui-ci dans les mêmes conditions que la

[4] Par *open source software*, on désigne différentes licences (GNU-GPL, LGPL, BSD, X Consortium, Artistic,..), toutes centrées sur les droits des utilisateurs-programmeurs [Perens (1998)]. La GNU-GPL est de loin la plus utilisée : 86 % des logiciels pris en compte dans la base WIDI [Robles et alii (2001)]. Le terme *free software* est parfois contesté ; le double sens du mot *free* en anglais (« libre », mais aussi « gratuit ») prêtant à confusion, car une éventuelle redistribution commerciale du code source fait partie des droits des utilisateurs.

[5] Il faut rappeler ici que le code écrit en vue d'une vente directe (sous forme de progiciel par exemple) n'est que la partie émergente d'un iceberg beaucoup plus étendu. La grande majorité du code (au moins 90 %) est écrite en interne pour des entreprises dont l'activité principale est autre (banques, sociétés d'assurances, etc.). Le travail

licence d'origine, et interdit d'incorporer un logiciel libre au sein d'un logiciel propriétaire. Ainsi, personne ne peut prendre aujourd'hui le contrôle légal du système Linux (sous GPL), car pour une privatisation éventuelle, il faudrait l'accord de milliers de contributeurs, ce qui est strictement impossible. Utilisant les lois du *copyright* en les détournant (*copyleft*), les licences *open source* représentent alors des dispositifs institutionnels élémentaires qui sécurisent à l'avance la production des logiciels libres et permettent d'installer dans la durée une certaine façon de produire le code et les programmes.

Trajectoires technologiques et institution : « GNU n'est pas Unix »

L'objectif du projet GNU (GNU's Not Unix) était la production d'un ensemble de logiciels sous licence GPL, comprenant un système d'exploitation compatible Unix (de manière à le rendre portable). Il s'agissait de « créer progressivement un fonds commun de logiciels libres dans lequel tout le monde pourrait puiser, auquel chacun pourrait ajouter, mais duquel personne ne pourrait retrancher » [Clément-Fontaine (1999)].

Les programmeurs qui vont travailler à ce projet héritent alors d'un ensemble de principes et d'outils (le langage C par exemple), qui définissent les architectures de type Unix : la stricte séparation entre le noyau (*kernel*) du système qui est seul à gérer les accès à la mémoire et au processeur (BIOS) et les différents processus utilisateurs ; des interfaces définies très précisément (standards) ; des programmes développés indépendamment les uns des autres, les différentes applications étant portables sur n'importe quel système d'exploitation Unix. Chaque logiciel de surcroît est construit comme un assemblage de fichiers (modules) ayant chacun une fonction particulière. Ces modules peuvent être écrits (et compilés) par morceaux, ce qui rend techniquement possible une construction progressive de l'ensemble.

L'histoire du projet GNU illustre parfaitement cette possibilité de débuter par n'importe quel élément, de traiter les différents éléments sans ordre, en produisant en dernier ce qui semble être la base du système (le noyau du système d'exploitation). « Au commencement, raconte Stallman [1998b], il s'agissait d'écrire n'importe quel composant, car tous les composants manquaient. L'ordre importait peu alors. Plus tard, une liste de ce qui manquait fut établie, la *task-list* de *GNU* ; cette liste permettait de recruter des développeurs pour telle ou telle partie du projet ». Vers 1990, presque tout le système existait, il manquait seulement le cœur du système d'exploitation, le noyau. La conception technique retenue par Stallman

---

des programmeurs est très souvent la maintenance d'un système informatique, une situation où celui qui écrit le code est aussi l'utilisateur direct de celui-ci.

(Le projet Hurd basé sur un principe de micro-noyau) s'avérant difficile à réaliser, c'est finalement un noyau compact de style classique écrit par Linus Thorvald en 1991 (Linux), qui a débouché le plus vite sur la production d'un système d'exploitation stable. Développé et diffusé sous GPL, il a permis de réaliser ce qui était le but même du projet GNU, un système complet de logiciels libres.

Ce qui sépare les programmeurs travaillant sur le projet GNU ou sur d'autres projets de même type (BSD par exemple), des autres programmeurs travaillant sur des logiciels propriétaires n'est alors ni les outils, ni les langages utilisés, ni même un comportement particulièrement innovateur ; la plupart du temps à l'époque, il s'agit simplement de libérer un programme qui existe déjà, un programme dont les principes et l'architecture sont connus [Narduzzo et Rossi (2003)]. La différence se situe plutôt au niveau des pratiques de partage et de réutilisation libre du code qui découlent du droit d'usage inscrit dans les licences. L'innovation institutionnelle permet alors de retrouver le processus d'invention collective qui caractérisait le développement des premiers systèmes d'exploitation Unix [Allen (1983) ; Nuvolari, (2003)]. Elle fonde une dynamique particulière d'accumulation, de circulation et de partage des savoirs dans le domaine des logiciels, qui donne aujourd'hui sa force aux différents projets *open source*.

II. *INSIDE THE BLACK BOX*, LES MYSTERES DU « CHAUDRON MAGIQUE »

On peut produire du logiciel libre dans le cadre d'une organisation traditionnelle, avec une équipe permanente salariée, une définition préalable de ce qui doit être produit, un plan, un calendrier et, le moment venu, la livraison du logiciel au client. Il n'y a rien dans les licences qui l'interdit, rien qui définit la manière dont on doit fabriquer le code. Ces licences, et la possibilité de partager, de modifier, de redistribuer librement, ont cependant donné naissance à de grands projets comme Apache, Linux, Gnome, Mozilla. Or ici, à la différence d'une organisation traditionnelle, il n'y a ni délais fixés à l'avance, ni répartition pré-établie des tâches, ni marché, ni hiérarchie, ni autorité découlant d'une relation salariale.

Mais alors, comment un produit commun peut-il apparaître ? Comment la coordination est-elle assurée ? Comment s'articulent la sphère marchande et la sphère non-marchande ? Comment en bref s'organisent le développement, la diffusion et l'évolution du logiciel dans ce type de projets *open source* ? Voilà quelques questions auxquelles nous allons essayer de répondre, en expliquant au préalable la manière dont on produit le logiciel dans des formes d'organisation plus traditionnelles.

Formes de production traditionnelles et cycle de développement du logiciel

Dans les projets mobilisant un grand nombre de programmeurs, le développement du logiciel est classiquement organisé comme une suite d'étapes successives : (a) conception, (b) écriture du code, (c) tests, (d) essais *in situ*, (e) maintenance. Mais l'expérience et les études de génie logiciel ont amplement démontré que ces étapes ne pouvaient être ni séparées, ni ordonnées, de manière trop rigoureuse.

Il y a d'abord un travail purement intellectuel de conception et d'écriture. Il faut définir le projet et son architecture, et ensuite mettre en forme le code-source. Celui-ci, une fois produit, doit être testé et corrigé. En effet, le code a beau être du texte, ce n'est pas un texte ordinaire. Il ne vaut que par son caractère fonctionnel et son efficacité technique. Mais il n'existe malheureusement pas de « preuve » en matière de code, et même une relecture minutieuse de celui-ci ne suffit pas à garantir que le code soit « bon », c'est-à-dire exempt de bogues ou d'erreurs, et conforme à ce que l'on souhaite obtenir. Le code est donc compilé et essayé sur machine, dans toutes les configurations possibles. La seule « preuve » possible est de nature empirique, et il faut toujours vérifier le code.

C'est la phase des tests. On lance toutes les commandes publiques, on essaye toutes les combinaisons raisonnables d'arguments[6]. Tous les défauts découverts doivent ensuite être corrigés en ré-écrivant le programme en tout ou en partie. Mais ces opérations peuvent introduire d'autres défauts, ce qui conduit à d'autres essais, et d'autres modifications. Lorsque le nombre de lignes de code augmente, la complexité du programme est telle que toutes les configurations possibles peuvent difficilement être vérifiées. Aussi, malgré des mois de tests intensifs, un programme complexe contiendra toujours des erreurs [Dréan (1996)]. Le développement prend donc la forme d'une suite d'opérations cycliques où l'écriture du code alterne avec des essais sur ordinateur, définissant un processus plus itératif que linéaire (qui semble d'ailleurs une parfaite illustration du modèle de Kline-Rosenberg [1991]).

Le développement du logiciel se prolonge dans des essais *in situ* faisant intervenir les différentes manières dont les utilisateurs finaux peuvent interagir avec les commandes du système informatique, puis dans des opérations de maintenance, qui là encore conduisent à retravailler le code-source : correction de bogues, ajouts d'extensions, etc. La maintenance est en effet cruciale pour la durée de vie d'un logiciel, qui sans elle deviendrait vite inutilisable. Le produit initial, même imparfait, est éternel. Il n'est jamais réellement consommé ou usé, et

[6] On écrit d'ailleurs pour cela des programmes spécifiques (code non fonctionnel). Cette activité représente entre 50 à 75 % du coût réel des projets informatiques [Printz (1998)].

ne s'abîme pas, sauf accident, quand on l'utilise. Mais sa durée de vie est limitée car l'environnement change. Les systèmes informatiques se modifient ; la demande des utilisateurs se transforme; le logiciel en conséquence doit être redéfini et réécrit.

Ecrire du code-source, le tester, corriger les défauts, sont des opérations qu'un programmeur individuel peut parfaitement réaliser. Ce travail ne demande pas de grands moyens techniques : du papier, un crayon, un traitement de texte élémentaire sont suffisants. Mais les projets logiciels comprenant un grand nombre de lignes de code sont difficilement réalisables sans la mobilisation et la coopération d'un grand nombre de programmeurs. Toutes les études et mesures de productivité ont alors montré :

(a) l'existence d'effets d'apprentissage importants ; la connaissance de l'historique du projet, la compréhension de l'architecture générale et de la place de chacun interviennent directement ici [Brooks (1996) ; Printz (1998) ; Horn (2000)]. Les productivités individuelles (mesurées conventionnellement en lignes de code-source par homme-mois) sont alors très variables (1 à 10 ou même 1 à 20, suivant les évaluations).

(b) le fait que la productivité globale est inversement proportionnelle à la taille d'un projet. La production des logiciels semble frappée de rendements d'échelle décroissants. Quand le nombre de personnes impliquées augmente, la productivité individuelle moyenne diminue. C'est la « loi de Brooks », du nom de celui qui, le premier, a constaté qu'ajouter des programmeurs à un projet qui prend du retard fait chuter brutalement la productivité de l'ensemble de l'équipe, et que le travail d'un programmeur pendant un an donnait un résultat bien supérieur au travail d'une équipe de douze programmeurs pendant un mois[7].

Comme l'inefficacité et le nombre de défauts augmentent avec la taille des programmes, il est toujours souhaitable quand cela est possible, de fractionner un programme en programmes de taille inférieure. Il faut alors que ces sous-programmes soient réellement indépendants, n'interagissant que par l'intermédiaire d'interfaces strictement définies et de petite taille. Ce constat a conduit à décomposer les grands systèmes en "modules", lesquels peuvent être écrits, testés et modifiés indépendamment les uns des autres ; un principe intégré aujourd'hui dans la définition de certains langages. Les systèmes Unix et le langage C reposent en particulier sur cette conception modulaire des logiciels.

[7] Et pourtant, "1 homme x 12 mois = 12 hommes x 1 mois" ! Mais l'équipe est moins efficace que l'individu. Les problèmes de communication et de coordination expliquent cet apparent paradoxe, dans une activité où l'action individuelle (écriture d'une ligne de code) peut interférer avec toutes les autres écritures.

Un mode de travail coopératif à l'écart de toute contrainte commerciale

Beaucoup de logiciels libres sont de petite taille et réalisés par un seul individu sans réelles contributions extérieures. Ainsi, l'examen de la base de données *sourceforge* fait apparaître que plus de la moitié des projets actifs et matures recensés sur cette base ont été produits par une seule personne. C'est le modèle de la « cave » [Krishnamurty (2002)]. D'autres projets à l'inverse mobilisent des équipes plus importantes (29 % impliquent plus de 5 développeurs) et concentrent alors la plus grande activité en termes de discussions. Même constat dans Orbiten [2000], à partir d'une base de données un peu différente. Le grand projet *open source*, mobilisant un grand nombre de contributeurs, n'est donc pas la forme dominante de production. Le système des licences a cependant rendu possible ce type de production. Rappelons-en les principales caractéristiques :

(a) Le (grand) projet *open source* mobilise un grand nombre de développeurs indépendants les uns des autres, lesquels en grande majorité sont des programmeurs professionnels. Les contributions ne sont pas rémunérées; les développeurs payés directement pour faire du logiciel libre étant minoritaires (20 % dans l'enquête WIDI [Robles et alii. (2001)]).

(b) Ce projet représente une forme de travail coopératif largement distribué géographiquement (plusieurs continents), les communications entre développeurs et le transport du code étant réalisés aux moindres frais grâce à l'internet.

(c) Le projet se construit et évolue indépendamment de toute pression commerciale et financière. Le but est le code lui-même et sa transformation ; les utilisateurs (programmeurs en général) étant souvent directement intéressés à voir le code-source s'améliorer[8].

(d) Le code source est de qualité variable, le code instable fréquent, et certains projets maintiennent plusieurs chemins de développement concurrents.

(e) Le projet se développe de manière cumulative et incrémentale et son rythme d'évolution est largement aléatoire. Les tests et la maintenance du code sont réalisés par un déboguage massif en parallèle, sans planning d'ensemble.

Cette forme de coopération dans le travail ne saurait exister sans les licences *open source*. Le système des droits de propriété (le *copyright* traditionnel) l'empêcherait. La coopération peut se mettre en place, se pérenniser, et déboucher sur des productions communes, parce que chaque auteur, propriétaire de certaines lignes de code, renonce à

[8] Toutes les enquêtes sur les motivations des contributeurs aux projets *open source* montrent la domination des raisons techniques : intérêt pour l'amélioration des fonctionnalités du programme ou formation personnelle (apprentissage), etc.

contrôler ce que deviendra sa propre production. Le système des licences élimine ainsi formellement tout conflit sur les usages du code et place d'emblée les programmeurs qui travaillent au même projet sur un pied d'égalité. Dans une organisation plus traditionnelle (une firme commerciale par exemple), c'est le rapport salarial qui, faisant de l'organisation la seule propriétaire du code, neutralise les effets destructeurs pour tout travail en coopération du système des droits de propriété individuels. Mais ici, le projet n'a nul besoin d'un rapport salarial ou d'un système de contrats, la micro-institution des licences *open source* suffit. Les relations qui se nouent autour du code sont alors foncièrement symétriques et égalitaires : tous ont les mêmes droits vis-à-vis du code : droits d'usage, de transformation, de redistribution.

La redistribution (et l'exploitation) commerciale du code est d'ailleurs possible : vente de *packaging* de logiciels libres, accompagnés ou non de logiciels propriétaires, activités de service utilisant directement le code source [Coris (2002)], etc. Les licences séparent alors deux activités pourtant liées : le travail sur le code, d'une part; et toutes les activités commerciales qui peuvent éventuellement apparaître en vue d'exploiter les produits du travail coopératif, d'autre part. Elles mettent ainsi en place une articulation originale et spécifique des sphères marchande et non-marchande[9].

Les projets *open source* reposent sur des formes indirectes de financement[10]. On a pu ainsi calculer le « coût » de certaines distributions de logiciels libres, en utilisant un modèle (Cocomo) qui permet d'évaluer, dans une structure propriétaire classique, le coût d'un projet en fonction de sa taille (en nombre de lignes de code). On obtient 1 892 millions de dollars pour Debian 2 .2 (14000 hommes-années) et 1000 millions de dollars pour RedHat Linux 7.1 (8000 hommes-années), des sommes impressionnantes [Gonzalez-Barahona et alii. (2002) ; Wheeler (2001)]. Mais ce calcul comparatif est évidemment fictif, ses sommes n'ayant jamais été ni réunies, ni réellement dépensées. Le mode de développement de l'*open source*, construit essentiellement à partir de contributions non rémunérées, permet en effet d'effectuer sans investissement la plus grande partie du travail; ce qui est particulièrement important au niveau du déboguage, un processus long et coûteux dans les modes propriétaires. La possibilité de réutiliser du code existant (même s'il s'agit d'un autre projet), ou de développer différents

[9] On pourrait comparer cette situation aux rues et aux routes financées par un budget public et que tous peuvent utiliser (et exploiter) librement, en installant tout autour leurs propres activités marchandes. Les logiciels commerciaux propriétaires correspondraient alors à la situation d'une concession autoroutière, avec des activités marchandes déléguées et contrôlées par le concessionnaire.

[10] Certaines firmes commerciales (Netscape, IBM, Sun, etc.) apportent leur code-source en le libérant, ou financent l'encadrement des projets (cas de Gnome ou Mozilla par exemple). Certains mainteneurs sont payés par des fondations (la FSF, etc.). Mais la plupart des contributions (sous forme de lignes de code) viennent de gens salariés, déjà payés donc pour écrire du code ou gérer des systèmes informatiques, sans que leur salaire soit directement lié au développement du projet *open source*.

projets en parallèle à partir du même code (toutes choses bien plus difficiles pour les développeurs d'un projet propriétaire, enfermés dans les « murs » de leur propre firme), réduisent d'ailleurs fortement l'effort total du développement.

La structure (émergente) d'une production distribuée, division du travail et hiérarchie

Dans les grands projets *open source*, l'activité s'organise sans organisateur apparent : il n'y a en principe ni plan à respecter, ni design prédéfini, ni délais d'aucune sorte, et chacun fait ce qu'il veut quand il veut. Et pourtant, toute une « communauté » de programmeurs coopère pour produire du code, de la documentation, des supports techniques, etc. A l'origine du projet, on trouve cependant toujours trois éléments : (1) un problème logiciel à résoudre, suffisamment important pour générer, le projet une fois lancé, la contribution d'un certain nombre de programmeurs, (2) un fondateur (personne ou équipe) suffisamment compétent pour prendre en charge la maintenance du projet, (3) un paquet de code, première matérialisation du futur programme, que d'autres développeurs peuvent tester, transformer, améliorer. Le projet est lancé par un appel à la communauté virtuelle des programmeurs, dont le développement de l'internet a même reculé les limites au monde entier. Cette communauté peut se mobiliser alors, au coup par coup ou de manière plus systématique.

Mais les projets *open source* sont loin pourtant de l'image véhiculée par le texte de Raymond [1998] sur « la Cathédrale et le Bazar », celle d'un simple réseau de pairs en interaction, dont Linux serait l'exemple phare. Ceux qui, une fois lancés, se développent effectivement, sont plutôt caractérisés par l'émergence d'une structure et d'une administration pratique du projet, ce qui veut dire division du travail (horizontale et verticale), répartition des rôles, et procédures de coordination. Les différents contributeurs ne sont alors ni strictement égaux, ni interchangeables. Certains ont un rôle plus important que d'autres et influent par leurs décisions sur le devenir du projet et sur les actions des autres contributeurs. Ils contrôlent même dans une certaine mesure le développement du projet, alors qu'ils sont effectivement dépourvus de tout moyen pratique de contraindre les autres à accepter leurs décisions. Le mystère n'est cependant pas très grand. On a affaire ici la plupart du temps à des contraintes et des problèmes techniques, et comme le dit Linus Thorvald : « Ce qui est agréable avec la technique, c'est que les problèmes techniques trouvent toujours une solution, alors que des questions non techniques n'ont parfois aucune réponse » [in Yamagata (1997)].

La coordination technique est réalisée par plusieurs dispositifs : l'usage systématique de la modularité, une administration du projet exercée par un (ou des) mainteneur(s), et un grand

nombre d'outils utilisant l'internet (CVS[11], forums (mails), système de suivi du traitement des bogues, etc.), ce qui assure une organisation routinière du travail d'ensemble, etc.

La modularité n'est pas l'apanage des projets logiciels *open source*[12]. Elle peut tout aussi bien exister dans des projets logiciels organisés de manière traditionnelle, et existe depuis longtemps dans les industries produisant des biens complexes comme l'industrie automobile, la construction aéronautique, etc. La modularité est ici un héritage des architectures Unix et du langage C, lequel est massivement utilisé ; il représente 70 % du code des distributions Debian [González-Barahona et alii (2002)]. En décomposant les gros programmes, la modularité rend possible l'établissement d'une division horizontale du travail et permet la spécialisation. On contourne ainsi la loi de Brooks en traitant les modules comme des boîtes noires, où la seule information disponible et révélée est l'interface [Narduzzo et Rossi (2003)]. Les différents modules peuvent être développés séparément sans qu'il soit nécessaire de coordonner le travail des programmeurs intervenant sur des modules différents. La cohérence d'ensemble est assurée par l'architecture générale du système, l'utilisation de bibliothèques, et le respect d'une définition précise pour les interfaces, une définition dont l'*open source* hérite pour partie (norme POSIX par exemple).

Un programme aussi important que Linux, 3 millions de lignes de code et 2300 développeurs identifiés en 1993, repose sur une telle architecture hiérarchisée de modules, leur nombre augmentant d'ailleurs d'une version Linux à l'autre [Gosh et David (2003)]. Il n'y a ici que quelques composants intégrés larges, comme le module de gestion de la mémoire (mm) ou le noyau lui-même (kernel module), les autres composants sont de taille bien inférieure. De tels modules sont développés par des équipes réduites (souvent un programmeur isolé[13]), et très nombreux sont les auteurs qui ne travaillent que sur quelques modules. Ainsi, plus de 70 % des auteurs de Linux n'ont travaillé que sur un seul module [Gosh et David (2003)]. Un programmeur seul peut facilement accéder à la compréhension d'un module tout entier, ce qui facilite le travail de ré-écriture et rend la

[11] Le répertoire central (CVS : *Current Version System*) reçoit et stocke les différentes contributions, le code accompagné par la documentation, les informations sur l'auteur, etc. Ce répertoire permet de retrouver les différentes versions et leur historique, et rend le code disponible pour toute la « communauté ».

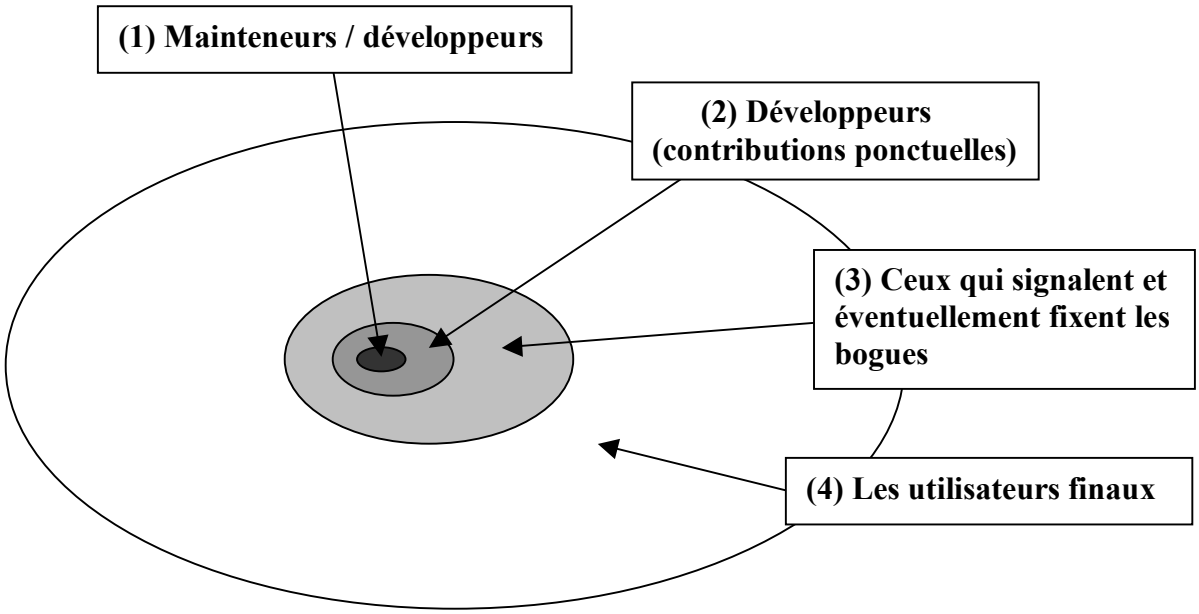
[12] C'est cependant une caractéristique de ce type de projets. Ainsi, pour mettre réellement sur pied le projet *open source* Mozilla à partir du code-source « libéré » du navigateur Netscape (1998), il a fallu introduire plus de modularité dans le code-source, afin de créer des conditions techniques qui permettent les contributions externes; la gestion de l'ensemble du projet étant assuré par un *staff*, composé d'ingénieurs payés par différentes entreprises (Netscape, Redhat, Oeone, Sun, IBM, etc.).

[13] Un constat général pour toutes les études qui ont tenté de mesurer la contribution des programmeurs à *l'open source*. Voir Orbiten [2000], FLOSS [Ghosh et alii (2002)], Krishnamurthy [2002], Healy et Schussman [2003],…

correction des bogues efficace. Contrairement alors à ce que véhicule l'image du « bazar », ce n'est pas l'intensité des communications dans une « communauté » indistincte, mais bien plutôt la suppression de tout besoin de communication qui fait la force des projets *open source*. Cette suppression est rendue possible par la modularité et permet l'apparition d'un travail géographiquement distribué.

Le deuxième trait caractéristique des projets *open source* est l'existence d'une division verticale du travail, avec une stratification marquée de la communauté des contributeurs. L'intensité des contributions est très variable et les rôles différenciés, ce qui ne peut exister de la même manière dans un projet organisé de manière traditionnelle, l'équipe (salariée) prenant alors en charge tout le projet. L'étude de Mockus, Fielding et Herbsleb [2000] sur Apache est caractéristique : 15 % des programmeurs impliqués dans le projet ont écrit 80 à 85 % du code. Et si plus de 3000 personnes ont bien travaillé sur le logiciel, c'est en fait une toute petite équipe qui en a assuré presque tout le développement, les autres ne contribuant que de manière secondaire, avec cependant une correction des bogues plus largement répartie. Il en est de même pour bien d'autres projets *open source*.

Le schéma général est alors celui d'un noyau de programmeurs assez stable (1), lequel contrôle le développement du code au niveau d'un module par exemple, en intégrant les contributions issues d'un groupe plus large, celui des développeurs qui contribuent ponctuellement au développement (2), entouré par ceux qui signalent ou fixent les défauts (bogues) (3), avec enfin la communauté beaucoup plus large des utilisateurs-finaux des versions (stabilisées) du logiciel (4). Le même individu peut appartenir à plusieurs de ces catégories.



Dans les projets de grande taille, la structure ci-dessus conduit à instaurer progressivement des formes organisationnelles et des outils spécifiques qui séparent clairement les rôles. Ainsi, l'ensemble du projet Apache est géré centralement par une équipe de mainteneurs qui décide de manière collégiale (en votant si nécessaire) de toute inclusion ou changement dans le code. Cette équipe, dont la taille a progressivement augmenté (8 personnes au départ, puis 12, puis 24), est composée de volontaires cooptés, chacun étant alors responsable d'un ensemble de modules.

La transformation du logiciel repose bien sur un principe de contributions libres, mais l'intégration des contributions à l'ensemble du code nécessite un certain filtrage, lequel est assuré par les mainteneurs responsables de tel ou tel module. Ainsi, dans le projet Mozilla [Reis et alii. (2002)], l'objectif est d'avoir à un moment donné une seule version du code. Le filtrage est alors hautement formalisé, avec la définition d'outils spécifiques qui assurent le suivi et l'organisation en temps réel du processus de révision (Bugzilla par exemple[14]). Dans ce projet, le terme *bug* désigne une demande de modification du logiciel, qu'il s'agisse d'un défaut actuel, d'une amélioration, de l'ajout d'une nouvelle fonctionnalité. Le *bug* est un ensemble de fichiers, comprenant les modifications proposées du code source (*patchs*), accompagnées d'un statut (non-confirmé, nouveau, attribué (à un responsable de module),

[14] Les premières années du projet Mozilla ont conduit à mettre au point différents outils (logiciels) nécessaires au bon fonctionnement des routines du travail distribué [Reis et alii (2002)]. Ainsi Bonsai, qui assure un accès direct (avec analyse) au répertoire CVS; et Bugzilla qui aujourd'hui est utilisé dans d'autres contextes, par la NASA, Connectiva, Redhat et GNOME, entre autres. Bugzilla enregistre les demandes de modification (*bug*) et leur traitement en temps réel, ce qui en fait un outil de communication et de coordination idéal pour les différents développeurs travaillant sur le projet.

résolu, réouvert, vérifié, clos), ce qui permet à chacun de savoir où en est le traitement du *bug*. Une fois enregistré, le *bug* va être l'objet d'une première revue (ligne à ligne), avec demandes de modifications éventuelles, suivie éventuellement d'une super-revue pour les introductions les plus problématiques, effectuée par un ingénieur plus expérimenté et plus familier du code. La même structure a progressivement émergée au cours du développement de Linux [Kuwabara (2000) ; Moon et Sproull (2000)] ; et, de ce point de vue, Linux apparaît même comme hautement hiérarchisé, puisqu'en principe, toute intégration de nouveaux *patchs* au code-source, doit être soumise à Linus Thorvald qui accepte, demande des modifications, ou rejette la contribution. Au fil du temps, une « liste de mainteneurs » responsables de tel ou tel module est apparue (de 3 à 147). Ce sont eux qui assurent le premier filtre et la première sélection en traitant les rapports sur les bogues, en définissant les nouveaux *patchs*, etc. ; et certains d'entre eux sont même devenus complètement autonomes dans leur domaine (comme Alan Cox), même si Linus Thorvald, qui garde la haute main sur le développement du noyau central du système d'exploitation (le module *kernel*) et sur les versions en développement, annonce toujours formellement les réalisations stables.

La base institutionnelle fait du partage et du libre usage du code la norme, mais ne peut définir dans le détail les formes organisationnelles et les routines nécessaires. On a là la marque de certaines contraintes techniques. La modularité permet un travail dispersé (et solitaire), mais l'intégration des différentes contributions reste un processus délicat. Pour assurer la qualité du code, le filtrage et un travail de réécriture parfois sont encore nécessaire. Le projet *open source* repose donc sur un système de murs filtrants : les contributions à l'entrée sont triées et examinées, avant d'être inscrites dans le code-source retenu, alors que la sortie est la plus ouverte possible ; n'importe qui pouvant venir télécharger librement le code des différentes versions en développement.

Un mode de production marqué par les cycles de développement et de vie du logiciel

Une différence importante entre le mode de production des logiciels propriétaires et le mode *open source*, réside dans le fait que le premier est conçu pour fabriquer des produits qui se veulent finis, alors que l'autre n'est qu'un processus de développement où le logiciel se transforme de manière cumulative et incrémentale et quasi-perpétuelle.

Pour les logiciels propriétaires, et particulièrement les progiciels, on a d'abord un investissement ordinaire, une avance en capital qu'il faut retrouver, accompagnée de profits les plus substantiels possibles lors de la vente des copies du produit fini. Ce mode de financement et la logique commerciale qui l'accompagne imposent leur loi à la production, car

on ne peut réellement vendre qu'un produit fini, une version stable débarrassée des bogues les plus importants, produite pour des utilisateurs-finaux qui n'interviendront pas sur le code. Cet impératif conduit à pousser le processus de développement du logiciel jusqu'à un certain point, et à interrompre celui-ci à partir de la livraison (quelques *updates* pouvant cependant servir à corriger les défauts découverts après le début de la commercialisation). Les délais de réalisation sont alors une contrainte cruciale pour le processus de développement et les investissements initiaux peuvent s'avérer importants (sans même parler des dépenses proprement commerciales), car on ne peut livrer un produit trop imparfait, et ces dépenses doivent nécessairement être amorties sur un volume minimal de ventes. Ceci étant réalisé, on peut entreprendre la production et la commercialisation d'une nouvelle version… en réutilisant éventuellement une partie du code propriétaire.

Il n'y a rien de semblable du côté d'un projet *open source*, car celui-ci n'est en principe qu'un projet de développement, la manifestation d'une certaine dynamique des connaissances techniques, débarrassée de toute entrave de nature commerciale. Alors que le système du logiciel propriétaire sépare producteurs et utilisateurs, le projet *open source* s'appuie au contraire sur les programmeurs-utilisateurs, lesquels sont très souvent directement intéressés à améliorer les performances du programme; l'exemple le plus probant étant ici Apache [Mockus et alii (2000) ; Von Hippel (2001)]. Alors que les pratiques commerciales imposent des délais, le respect d'un planning, un rythme bien régulier, ce qui suppose un management spécifique, et impose un processus du développement qui a un début et une fin, le projet *open source* commence bien à un certain moment du temps, mais ne connaît ni délais, ni rythme bien défini, ni fin programmée. Au départ, il n'est même pas besoin d'avoir une définition précise de ce que doit être le logiciel. Il faut plutôt un bon paquet de code-source qui intègre en lui-même (avec la documentation) une architecture et une vision générale de ce qu'on veut obtenir, laquelle va guider le travail des différents développeurs.

On a donc affaire à un processus évolutionniste organisé, combinant accumulation et sélection. Les changements dans le code-source se font dans une succession d'ajouts et de suppressions de paquets de lignes de code. Le résultat est la croissance et la transformation de l'ensemble. Le code change ainsi en fonction des problèmes que rencontrent les utilisateurs, qu'il s'agisse simplement de corriger des défauts, d'introduire de nouvelles fonctionnalités ou de redéfinir une partie de l'architecture du programme, sans qu'il soit nécessaire d'établir à l'avance un plan de travail général. Les pratiques de filtrage et l'intervention des mainteneurs suffisent à conserver et améliorer la qualité du code et la cohérence d'ensemble.

Un tel processus ne semble pas toujours capable par lui-même de délivrer les produits finis dont les utilisateurs-finaux ont besoin. On voit alors souvent apparaître deux stockages et deux distributions de code source, destinées l'une aux utilisateurs-programmeurs, l'autre aux utilisateurs finaux. Ainsi depuis 1994, Linux distingue les noyaux stables, pour lesquels on cherche seulement à corriger les erreurs (bogues fixes) sans toucher aux algorithmes et aux structures de données, et les noyaux en développement, où on peut expérimenter assez librement de nouvelles solutions et introduire des changements plus fondamentaux[15]. Différentes versions peuvent ainsi être développées en parallèle, et donner ensuite différentes versions stables. Une telle structure permet à la fois la pérennité du processus de développement et l'amélioration (par simple maintenance) des produits finis, en donnant aux utilisateurs finaux des versions fiables et directement utilisables.

Les rares analyses de l'évolution interne des logiciels *open source* ont alors montré que les différents modules n'évoluaient pas au même rythme et de la même façon, la nature du module étant déterminante ici. Si la taille de Linux a progressé en quelques années de manière significative, c'est essentiellement le fait du fichier *drivers* (60 % des lignes de code), avec l'augmentation du nombre des pilotes d'extensions, une évolution directement tirée par la demande des utilisateurs et la popularité croissante de Linux[16]. Il en est de même pour le fichier *arch*, stockant le code nécessaire pour établir la portabilité sur différents matériels lors de la compilation. D'autres fichiers par contre n'ont guère changé (les librairies, depuis 1995), ainsi que le cœur du système, le noyau proprement dit (*module kernel*), et le module qui gère la mémoire (*mm*). Même constat pour Gnome, un projet GNU lancé en 1996 en vue de créer un environnement bureau accompagnée d'une suite Office ; certains modules (gnome-core, gtk, ORBit) semblent avoir atteint leur phase de maturité, les seules interventions étant motivées par la correction de défauts mineurs [Koch et Schneider (2000)].

Il apparaît donc que, dans ces formes de production et de développement *open source*, les contributions libres et apparemment désorganisées des différents développeurs dispersés

[15] On utilise un système de numérotation simple, pour toutes les nouvelles publications. Le code 2.3.51 donne le numéro de version (2) et le numéro de publication (51), le nombre 3 (impair) signifiant une version en développement. Le code 2.2.14 correspond à une version stable, le nombre 2 étant pair. Les distributions Debian d'une manière analogue distinguent la « patate » (la réalisation officielle stable du moment), et les réalisations en préparation, « woody » et « sid », suivant le degré d'instabilité et de traitement des bogues [Gonzalez-Barahona et alii (2002)].

[16] Au total, la croissance du noyau Linux s'avère impressionnante. Une comparaison entre les différentes versions (de mai 1994 à juillet 2002) de Linux fait apparaître l'augmentation du nombre d'auteurs (de 158 à plus de 2000) et du nombre de lignes de code (de 122 000 à plus de 3 millions de lignes) [Ghosh et David (2003)]. La comparaison statistique est cependant trompeuse, car le logiciel n'est pas un produit manufacturé, avec des modèles différents qui doivent être (re)fabriqués de a jusqu'à z, mais un seul produit perpétuellement retravaillé et ré-écrit, qui croît donc par modifications et accumulations successives.

dans le monde sont assez largement ordonnées par des déterminismes d'origine technique ou les pressions d'une demande directe - hors marché - des utilisateurs (souvent eux-mêmes programmeurs).

CONCLUSION

Les logiciels sont parfois des marchandises, mais plus couramment et avant tout des valeurs d'usage. Comme valeurs d'usage, ce sont des sortes d'outils qui ne sont pas recherchés en vue d'une « consommation finale », mais bien plutôt pour ce qu'ils permettent de faire et de produire. Constitués à partir d'une écriture (le code-source), ils se distinguent clairement de ces autres formes d'écritures que sont par exemple les romans ou les poèmes. Les usages sociaux en sont bien différents ; et l'usage d'un outil logiciel implique presque naturellement sa ré-écriture, sa modification, ou sa transformation en vue d'une autre utilisation.

Ces outils sont d'ailleurs facilement copiables, mais en même temps, leur duplication éventuelle, même en très grand nombre, ne les fait ni disparaître, ni n'en dépossède les premiers utilisateurs. Ce point a conduit, dans le cadre du *copyright* et dans une logique de protection du *fructus* à fermer l'accès au code-source et à interdire toute pratique de partage et de ré-utilisation du code. C'est à l'inverse pour favoriser et sécuriser ces pratiques que sont apparues, par détournement du système du *copyright*, les licences *open source*, qui reposent sur une définition particulièrement extensive des droits d'usage concédés par le propriétaire en titre, une définition taillée sur mesure pour les utilisateurs-programmeurs.

Dans la continuité alors des premières formes de coopération « libres » et distribuées autour du système Unix, un fond commun de logiciels et de code *open source* a été progressivement constitué et s'étend de jour en jour, un fond dans lequel tout le monde peut puiser, à la seule condition d'accorder aux autres les mêmes droits d'usage. Il y a là une réorganisation micro du système des droits de propriété qui : (1) sacrifie le *fructus* à l'*usus*, (2) fait du partage du code la norme, et donne ainsi naissance à une sorte de propriété commune, (3) sépare activité commerciale et activité de production, et modifie ainsi profondément les conditions de développement des projets logiciels.

C'est sans doute la raison la plus fondamentale du succès de ces licences. Elles ont permis, en dehors de toute contrainte et pression commerciale, de développer et de faire grandir de manière incrémentale des logiciels dans un rapport constant entre utilisation et production. Elles ont aussi permis l'émergence de formes sociales de travail coopératif distribué à l'échelle planétaire. Les grands projets *open source* sont alors apparus, un

mouvement largement tiré par une demande non monétaire directement exprimée en termes de valeurs d'usage. La force du « chaudron magique » est alors pour l'essentiel celle de l'invention collective et de la coopération volontaire, la circulation des connaissances et du code échappant alors à la sphère marchande.

Références :

Allen, R. C. [1983], « Collective Invention », *Journal of Economic Behavior and Organization*, 4, p. 1-24.

Brooks, F. P. [1996], *Le mythe du mois-homme. Essais sur le génie logiciel*, International Thomson Publishing, Paris.

Bessen, J., Maskin, E. [1999], « Sequential Innovation, Patents, and Imitation », *Working paper*, http://researchinnovation.org/patent.pdf.

Bezroukov, N. [1999a], « Open source software development as a special type of academic research (critique of vulgar Raymondism) », *First Monday*, 4(10), October.

Bezroukov, N. [1999b], « A second look at the Cathedral and Bazaar », *First Monday*, 4(12), December.

Browne, C. B. [1998], « Linux and Decentralized Development », *First Monday*, 3 (3), March.

Clément-Fontaine, M. [1999], *Une étude juridique de la Licence Publique Générale GNU*, mémoire de DEA en Droit, http://www.crao.net/gpl/

Coris, M. [2002], « Les sociétés de services en logiciels libres : l'émergence d'un système de production alternatif au sein de l'industrie du logiciel », Université Montesquieu, Bordeaux, IFREDE-E3I, *Document de travail*, n° 2002-1.

DiBona, C., Ockman, S., Stone, M. (eds) [1999], *Open sources – Voices from the open source revolution*, O'Reilly & Associates, Sebastopol, California.

Dempsey, B. J., Weiss, D., Jones, P., Greenberg, J. [1999], « A quantitative profile of a community of open source Linux developers », *Working paper*, http://netalab.unc.edu/osrt/develpro.htm.

Dréan, G. [1996], *L'industrie informatique. Structure, économie, perspectives*, Masson, Paris.

Foray, D., Zimmermann, J.B. [2001], « L'économie du logiciel libre. Organisation coopérative et incitation à l'innovation », *Revue Economique*, 52, numéro hors série, p.77-93.

Ghosh, R. A., Glott, R., Krieger, B., Robles, G. [2002], « FLOSS : Free/Libre/Open Source Software Study », MERIT, http://floss.infonomics.nl/report.

Ghosh, R. A., David, P. [2003], « The nature and composition of the Linux kernel developer community, a dynamic analysis », http://dxm.org/papers/licks1/licksresults.pdf.

Gnoff, J. et Weinberg, P. [1989], *Unix, une approche conceptuelle*, InterEditions, Paris.

Godfrey, M. W., Tu, Q. [2000], « Evolution in Open Source Software: A Case Study », *Proceedings ICSM*, http://plg.uwaterloo.ca/~migod/papers/icsm00.pdf .

González-Barahona, J., Ortuno-Perez, M., Quirós, H., Centeno González, J., Matellán Olivera, V. [2002], « Counting potatoes : the size of Debian 2.2 », *Working paper*, http://opensource.mit.edu/papers.

Healy, K., Schussman, A. [2003], « The ecology of open-source software development », http://opensource.mit.edu/papers.

Horn, F. [2000], *L'économie du logiciel*, Thèse de Doctorat en Sciences Economiques.

Horn, F. [2001], « La diversité de l'économie du logiciel : pluralité et dynamique de quatre « mondes de production » », *Revue d'Economie Industrielle*, 95, p. 37-60.

Horn, F. [2004], *Economie du logiciel*, La Découverte, Paris.

Jullien, N. [1999], « Linux, convergence du monde UNIX et du monde PC ? », *Terminal*, n° 80-81, http://www.terminal.sgdg.org/no_speciaux/80_81/Jullien.html.

Jullien, N., Zimmermann, J.B. [2001], « Le logiciel libre : une nouvelle approche de la propriété intellectuelle », GREQUAM, *Document de travail,* n° 01B06.

Kline, J. et Rosenberg, N. [1991], « An overview of innovation », in Landau et Rosenberg (éds.), *The Positive Sum Strategy*, National Academic Press, Washington DC.

Koch, S., Schneider, G. [2000], « Results from software engineering research into open software development projects using public data », *Working paper*, http://opensource.mit.edu/papers.

Krishnamurthy, S. [2002], « Cave or community ? An empirical examination of 100 mature open source projects », *Working paper*, http://faculty.washington.edu/sandeep.

Kuwabara, K. [2000], « Linux: a Bazaar at the edge of chaos », *First Monday*, 5(3), march.

Lerner, J., Tirole, J. [2002], « Some simple economics of open source », *Journal of Industrial Economics*, 50, p. 197-234.

Mockus, A., Fielding, R.T., Herbsleb, J. [2000], « A case study of open source software development : the Apache server », *Proceedings of ICSE'2000*.

Moon, J.M., Sproull, L. [2000], « Essence of distributed work: the case of the Linux kernel », *First Monday*, 5(11), november.

Mowery, D. C. et Rosenberg, N. [1998], *Paths of innovation, technological change in 20th-century America*, Cambridge University Press.

Narduzzo, A. et Rossi, A. [2003], « Modularity in Action : GNU/Linux and Free/Open Source Software Development Model Unleased », *Working paper*, http://opensource.mit.edu/papers.

Nuvolari, A. [2003], « Open source software development : some historical perspectives », *Working paper*, http://opensource.mit.edu/papers.

Orbiten [2000], *The Orbiten free software survey*, *first edition*, http://orbiten.org/ofss/01.html.

Perens, B. [1998], « La définition de l'open source », in *DiBona et alii*. [1999], disponible sur http://perens.com/Articles/OSD.html.

Printz, J. [1998], *Puissance et limites des systèmes informatisés*, Hermes, Paris.

Raymond, E. [1998], « The Cathedral and the Bazaar », *First Monday*, 3 (3), march.

Raymond, E. [1999], « The magic cauldron », in *The Cathedral & the Bazaar*, O'Reilly & Associates, Sebastopol, Californie, trad.fr., http://www.linux-france.org.

Reis, C.R., Pontin de Mattos Fortes, R. [2002], « An overview of the software engineering process and tools in the Mozilla project », *Working paper* , http://opensource.mit.edu/papers.

Robles, G., Scherder, H., Tretkowski, I., Weber, N. [2001], « WIDI : Who is doing it ? A research on libre software developers », *Working paper* , http://widi.berlios.de/paper/study.html.

Rosenberg, N. [1982], *Inside the Black Box, Technology and Economics*, Cambridge University Press.

Salus, P. H. [1994], *A Quarter Century of UNIX*, Addison-Wesley Publishing Company, Inc, New York.

Shapiro, C., Varian, H. [1999], *Economie de l'information, guide stratégique de l'économie des réseaux*, De Boeck Universités, Bruxelles.

Stallman, R. [1998a], « Le système d'exploitation GNU et le mouvement du logiciel libre », in *DiBona et alii.* [1999].

Stallman, R. [1998b], « Conférence donnée à Paris VIII », http://www.linux-france.org.

Unix Berkeley [1999], « Deux décennies d'Unix Berkeley – de la version AT&T à la version libre », in *Di'Bona et alii.* [1999], trad. fr., http://www.editions-oreilly.fr/divers/tribune-libre/fr-x489.html.

Von Hippel, E. [2002], « Open source projects as horizontal innovation networks – by and for users », *MIT Sloan School of Management Working Paper*, n° 4366-02.

Von Hippel, E., Lakhani, K. R. [2000], « How open source software works : "free" user-to-user assistance », *MIT Sloan School of Management Working Paper*, n° 4117-00.

Von Hippel, E. [2001], « Innovation by users communities : learning from open source software », *MIT Sloan Management Review*, 42(4), p. 82-86.

Von Hippel, E. [2002], « Open source projects as horizontal innovation networks – by and for users », *MIT Sloan School of Management Working Paper,* n° 4366-02, juin..

Wheeler, D. A. [2001], « More Than a Gigabuck: Estimating GNU/Linux's Size », http://www.dwheeler.com/sloc.

Wilson, L. B. et Clarck, R. G. [1993], *Langages de programmation comparés*, Addison-Wesley.

Yamagata, H. [1997], « Le pragmatiste du logiciel libre : entretien avec Linus Torvalds », trad. fr. Blondeel, sur http://www.linux-france.org.

Zimmermann, J.B. [1999], « Logiciel et propriété intellectuelle : du Copyright au Copyleft », *Terminal*, n°80/81, Editions L'Harmattan, p.151-166.